Optimal number of threads in parallel computing

Pavel Kazenin, www.linkedin.com/in/pavelkazenin/ , pavel.kazenin@gmail.com

Abstract

In this article, a method for determining the optimal number of processing threads in multithreaded programs is described and experimentally verified for quick sort algorithm.

Introduction

One of the most common problems a software developer faces when developing multithreaded program is choosing optimal number of simultaneously running threads processing a data set. Increasing the number of threads reduces execution time of each thread but increases an overhead imposed by switching application context between threads and possible pre- and post-processing operations.

In this article a simple method is suggested to estimate the optimal number of threads n_{min} by making two measurements for two different arbitrary number of threads *m* and *n*. Theoretical results are then verified for parallel quick sort. Implementation of the algorithm and test programs can be found on GitHub.

General equations and formulas

Execution time to process volume *v* of data by *n* threads can be expressed as

$$T(v,n)=T_n=X(\frac{v}{n})+Y(vn)+V(n)+Z(v)$$

where *X(p)* is execution time of each thread, *Y(p)* is overhead caused by multithreading and post-processing of intermediate results, *V(n)* is time needed to set up threads before and clean them up after data processing. We can ignore *V(n)* as it is small comparing to *X* and *Y for* $v \gg n$ *: V(n) ≈ 0.* Function *Z(p)* is independent of *n* and can be determined from boundary conditions:

$$T(v,1)=X(v) \;\rightarrow\; Y(v\cdot 1)+Z(v)=0 \;\rightarrow\; Z(v)=-Y(v)$$

Final formula for $T(v,n)$ can then be written as follows:

$$T(v,n)=T_n=X(\frac{v}{n})+Y(vn)-Y(v) \qquad (1)$$

The optimal number of threads n_{min} which satisfies the criteria $T(v, n_{min}) = min(T(v, n))$ is a solution of partial differential equation $\partial T(v, n)/\partial n = 0$:

$$\frac{\partial T(n,v)}{\partial n} \;=\; \frac{\partial}{\partial n}\left[X\left(\frac{v}{n}\right) + Y(vn) - Y(v) \right] \;=\; -\frac{v}{n^2} X' + vY' \;=\; 0$$

where *X′* and *Y′* are derivative functions of *X* and *Y*. By solving this equation for *n*, we derive equation for n_{min} :

$$n_{min}^2 \;=\; \frac{X'\left(\frac{v}{n_{min}}\right)}{Y'(v\, n_{min})} \qquad (2)$$

Functions *X(p)* and *Y(p)* are proportional to their corresponding big-O complexity functions *x(p)* and *y(p)*:

$$X(p) = X_0 x(p); \quad Y(p) = Y_0 y(p)$$

$$X'(p) = X_0 x'(p); \quad Y'(p) = Y_0 y'(p)$$

We can calculate coefficients X_0 and Y_0 using measurable parameters T_m and T_n :

$$\left\{ \begin{array}{l} T_m = X_0 x\left(\frac{v}{m}\right) + Y_0 y(vm) - Y_0 y(v) \\ T_n = X_0 x\left(\frac{v}{n}\right) + Y_0 y(vn) - Y_0 y(v) \end{array} \right\} \rightarrow \left\{ \begin{array}{l} X_0 = \dfrac{T_n(y(vm) - v(v)) - T_m(y(vn) - v(v))}{x\left(\frac{v}{n}\right)(y(vm) - v(v)) - x\left(\frac{v}{m}\right)(y(vn) - v(v))} \\ Y_0 = \dfrac{T_m x\left(\frac{v}{n}\right) - T_n x\left(\frac{v}{m}\right)}{x\left(\frac{v}{n}\right)(y(vm) - v(v)) - x\left(\frac{v}{m}\right)(y(vn) - v(v))} \end{array} \right\}$$

Substituting the coefficients into (2), we derive final equation for n_{min}

$$n_{min}^2(v) = \frac{T_m(y(vn) - y(v)) - T_n(y(vm) - y(v))}{T_n x\left(\frac{v}{m}\right) - T_m x\left(\frac{v}{n}\right)} \cdot \frac{x'\left(\frac{v}{n_{min}}\right)}{y'(v\, n_{min})} \qquad (3)$$

Root function of this equation $n_{min}(v)$ is the optimal number of threads to process volume *v* of data.

Parallel quick sort

For parallel quick sort and post-sort linear merge of the results, the complexity functions are $x(p)=p\ln p$ and $y(p)=p$, respectively, and *X(p)* and *Y(p)* and their derivatives can be written as follows:

$$X(p)=X_0 p\ln p;\;\; Y(p)=Y_0 p$$

$$X'(p)=X_0(\ln p+1);\;\; Y'(p)=Y_0$$

Substituting the above formulas into (3), we derive equation for n_{min} :

$$n_{min}^2=\frac{mn(T_m(n-1)-T_n(m-1))}{T_n n\ln(\frac{v}{m})-T_m m\ln(\frac{v}{n})}\cdot(\ln(\frac{v}{n_{min}})+1) \qquad (4)$$

There is no exact analytical solution of this equation, however the equation can be solved numerically:

$$n_{min}^2=-A\ln n_{min}+B \qquad (5)$$

where

$$A=\frac{mn(T_m(n-1)-T_n(m-1))}{T_n n\ln(\frac{v}{m})-T_m m\ln(\frac{v}{n})};\quad B=A(\ln v+1)$$

See Appendix A for numeric solution of equation (5).

In case of very large data volume $v\gg n\approx m$ and $\ln v\gg\ln n\approx\ln m\approx 1$, $n_{min}=const(v)$ and

$$n_{min}\approx\left(\frac{mn(T_m(n-1)-T_n(m-1))}{T_n n-T_m m}\right)^{\frac{1}{2}} \qquad (6)$$

Multi-dimentional parallel computing

For multi-dimentional computing algorithms, the algorithm complexity can be written as power function of degree *k*: $x(p)=p^k$. For example, bubble sort of random array has complexity of $x(p)=p^{1.5}$. For such algorithms, functions *X(p)* and *Y(p)* and their derivatives can be expressed as

$$X(p)=X_0 p^k; \quad Y(p)=Y_0 p$$

$$X'(p)=X_0 k p^{k-1}; \quad Y'(p)=Y_0$$

Substituting the above formulas into (3), we derive equation for n_{min} :

$$n_{min}^2=\frac{T_m v(n-1)-T_n v(m-1)}{T_n(\frac{v}{m})^k-T_m(\frac{v}{n})^k}\cdot k(\frac{v}{n_{min}})^{k-1}$$

After all reductions, we get final formula for n_{min} :

$$n_{min}=\left(\frac{m^k n^k k(T_m(n-1)-T_n(m-1))}{T_n n^k-T_m m^k}\right)^{\frac{1}{k+1}} \qquad (7)$$

Note that n_{min} does not depend on volume *v*, meaning we don't have to wait for single threaded and multithreaded processes to complete the whole data array, instead, we measure T_m and T_n for a portion of the entire data set.

Another interesting fact is that for $k=1$, that is, linear complexity *x(p)=p*, the formula reduces to expression (6) we previously derived for quick sort in case of $v \gg n \approx m$:

$$n_{min}\approx\left(\frac{m n(T_m(n-1)-T_n(m-1))}{T_n n-T_m m}\right)^{\frac{1}{2}}$$

This is because for huge volume *v*, $O(v \ln v)\approx O(v)$, in other words, for large data array, the complexity of quick sort algorithm becomes linear.

Experimental results for Parallel Quick Sort algorithm

n	1	2	3	4	6	8	11	16	23	32	45	64	91	128	181
T_n	1249	671	546	437	452	469	546	686	890	1155	1514	1966	2652	3572	4914
n_{min}	4.01	4.07	4.01	4.35	4.75	5.02	5.13	5.37	5.57	5.83	5.84	6.18	6.35	6.45	6.54

Table 1. Execution time in milliseconds and calculated optimal number of threads as a function of number of threads, *m=1*.

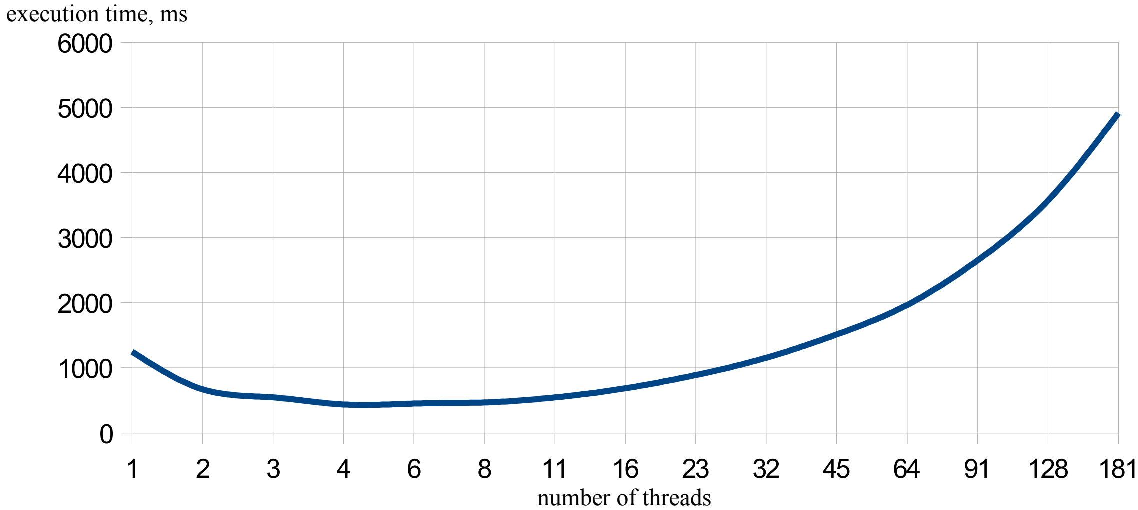

Figure 1. Execution time for parallel quick sort. Data set is a random array of long integers, volume $v = 10^7$.

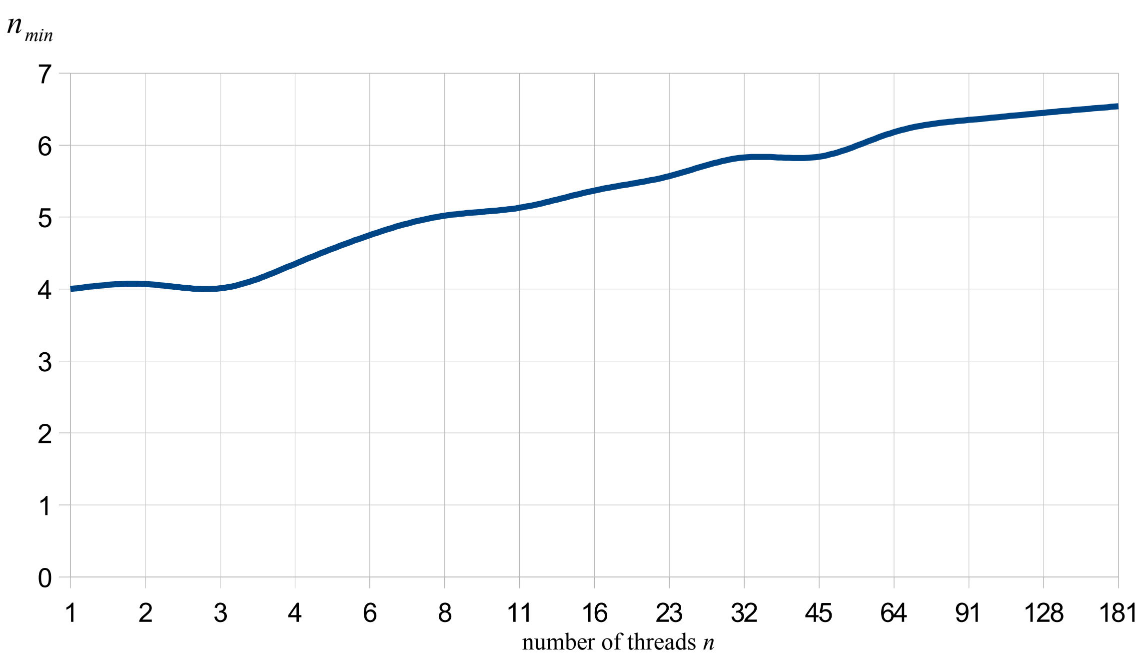

Figure 2. Theoretical n_{min} for quick sort, *m=1*. Data set is a random array of long integers, volume $v = 10^7$

Appendix A. Numeric solution of $x^2 = -A\ln(x) + B$

Below is implementation of numeric solution of the equation (5) in Java programming language. Root of the equation is the optimal number of threads for parallel quick sort.

```
/*
 *  Numeric solution of equation x^2 = -a*ln(x) + b.
 *  The root of the equation is the optimal number of threads for parallel quick sort.
 *
 *  @param   v          volume of data set
 *  @param   m          first measured number of threads
 *  @param   n          second measured number of threads
 *  @param   tm         execution time for m number of threads
 *  @param   tn         execution time for n number of threads
 *  @return  nOptimal   optimal number of threads for parallel quick sort or -1
 *  @throws  IlligalArgumentException when v<=0 or m<=0 or n<0 or tm<=0 or tn<=0 or m == n
 */
public static int calculateOptimalThreadsQuickSort (
        int v, int m, int n, long tm, long tn) throws IllegalArgumentException {

            if (v <= 0)  throw new IllegalArgumentException("illegal arguments, v <= 0");
            if (m <= 0)  throw new IllegalArgumentException("illegal arguments, m <= 0");
            if (n <= 0)  throw new IllegalArgumentException("illegal arguments, n <= 0");
            if (tm <= 0) throw new IllegalArgumentException("illegal arguments, tm <= 0");
            if (tn <= 0) throw new IllegalArgumentException("illegal arguments, tn <= 0");
            if (m == n)  throw new IllegalArgumentException("illegal arguments, m == n");

            double x = 1;

            double f1, f2, df1, df2;
            double a = (n*m*(tm*(n-1)-tn*(m-1)))/(tn*n*Math.log(v/m)-tm*m*Math.log(v/n));
            double b = a*(Math.log(v)+1);
            double delta = 1;
            double sigma = 1;

            for (int i=1; Math.abs(sigma) > 0.00001 && i<100; i++) {

               f1    = x*x;
               df1   = 2*x;
               f2    = -1*a*Math.log(x)+b;
               df2   = -1*a/x;

               delta = (f1-f2)/(df1-df2);
               x     = x-delta;
               sigma = delta/x;
        }

        return (x == Double.valueOf(Double.NaN) ? -1 : (int)Math.ceil(x));
}
```